Proposal for an ILC TPC data stream

Ties Behnke[a], Maximilien Chefdeville[b],
Frank Gaede[a], Christian Hansen[c], Matthias Enno Janssen[a],
Alexander Kaoukher[d], Martin Killenberg[e],
Jason McGeachie[c], Astrid Münnich[e], Adrian Vogel[a],
Michael Weber[e], Peter Wienemann[f]

[a] *DESY*

[b] *NIKHEF*

[c] *University of Victoria*

[d] *University of Rostock*

[e] *RWTH Aachen*

[f] *University of Freiburg*

Draft from July 3, 2006

Abstract

This document proposes a TPC data flow model for use during ILC detector R&D studies. It is based on LCIO data structures and Marlin as analysis and reconstruction framework.

1 Introduction

At present there is a large diversity of simulation, reconstruction and analysis software available within the ILC TPC community. Typically every group uses its own data format which makes the exchange of data and code time consuming and errorprone (e. g. due to the usage of different coordinate systems). If the limited personpower would be focussed on the development of a software framework shared among the groups, the quality of the algorithms would improve significantly faster and the comparison of results would be simpler.

Due to all the different readout structures, electronics, amplification system, etc. used by the ILC TPC group, such a common framework needs to be highly modular, such that only a small part of code needs to be adapted for different setups.

To standardize the exchange of data within and among software frameworks, the LCIO (linear collider I/O) persistency framework [1] was developed as a data model for linear collider detector studies. It offers data structures for all stages of event reconstruction, starting from detector raw data to fully reconstructed objects. In addition to that simulation specific data can be stored. Since LCIO (by intention) leaves some freedom to the user, this document proposes conventions to make sure that a certain class always stores the data using the same units, same coodinate system, etc. This is indispensable to avoid confusions during the exchange of data.

Continuing this idea one step further would include the exchange of simulation, reconstruction and analysis code. A modular reconstruction and analysis framework based on LCIO is Marlin [2]. It has been designed in such a way that distributed development of modules, so called processors, is as simple and straight forward as possible. Each processor (operating on an event-by-event basis) executes one step in the reconstruction or analysis chain. To work properly the reconstruction code needs geometry information, calibration constants, electronics properties, etc. To make sure that all processors make use of a consistent set of parameters, these numbers are obtained through GEAR [3] and LCCD [4]. GEAR is used to store static information (pad geometry, readout frequency, etc.) whereas LCCD is used to save conditions data which can change during data taking (drift velocity, voltages, B field, calibration constants, metrological data, etc.).

This document is organized as follows: Section 2 will describe in detail the tracking related classes of LCIO. In Section 3 the Marlin processors are proposed for a step-by-step reconstruction of the events. Classes to access conditions data from LCCD are explained in Section 4. The last part is committed to the usage of the described classes for non-standard readout technologies like TDC based or pixel based readouts. **FIXME: IS THIS STILL UP-TO-DATE???**

2 LCIO classes

This section is devoted to the discussion of the TPC related classes in LCIO.

2.1 `TrackerRawData`

This class is only needed for TPCs with FADC readouts. The output of the readout electronics goes into objects of this class, i. e. it contains raw data without any correction applied.

- `int getCellID0()`
  returns the hardware ID of channel assigned by electronics.
- `int getCellID1()`
  not used for the TPC.

- `int getTime()`
  returns the FADC time bin number (counting starts from 0) of the first FADC bin contained in the collection returned by `getADCValues()`.

- `const ShortVec& getADCValues()`
  returns contents of the FADC time bins in FADC counts.

2.2 TrackerData

This class is only needed for TPCs with FADC readouts. Objects of this type contain data with all *electronics* related corrections applied (gain variations are not corrected for yet).

- `int getCellID0()`
  returns the hardware ID of channel assigned by electronics.

- `int getCellID1()`
  not used for the TPC.

- `float getTime()`
  returns the time (in units of FADC time bins) to the left edge of the first FADC bin contained in the collection returned by `getADCValues()`. This already includes corrections for possible time offsets (therefore the return type `float`).

- `const FloatVec& getChargeValues()`
  returns contents of the FADC time bins in FADC counts after applying all *electronics* related corrections like pedestal substraction, correction for non-linear FADC behaviour etc. (gain variations are not corrected for yet).

2.3 TrackerPulse

This data structure is used to store reconstructed pulses in the individual channels. Here all corrections are applied (even those for gain variations).

- `int getCellID0()`
  returns the software ID of the channel.

- `int getCellID1()`
  not used for the TPC.

- `float getTime()`
  returns the reconstructed time of the pulse in ns.

- `float getCharge()`
  returns the reconstructed total charge (time integral) of the pulse in units of primary electrons.

- `int getQuality()`
  returns quality bits for the pulse. If no bits are set, the pulse looks normal. If bits are set (the counting is such that the least significant bit has number 0), they have the following meaning:

  - bit 0: pulse is double pulse candidate.
  - bit 1: pulse originates from a double pulse candidate which has been split up into single pulses.
  - bit 2: anomalous pulse shape.
  - bit 3: underflow in at least one time bin.
  - bit 4: overflow in at least one time bin.
  - bit 5 - 19: not used for the time being.
  - bit 20 - 30: reserved for private usage.
  - bit 31: pulse spectrum storage forced for more detailed studies.

  To test whether bit `n` has been set, you can e. g. use the following code snippet:

  ```
  if ( pulse.getQuality() & ( 1 << n ) ) {
      // ...
  }
  ```

- `TrackerData* getTrackerData()`
  Pointer to corresponding `TrackerData` object. This pointer is only set if `pulse.getQuality() != 0`.

2.4 TrackerHit

Class to store "hits" along a track. This can be per-row information for conventional pad readouts, cluster data for pixel readouts or similar information.

- `const double* getPosition()`
  `getPosition()[0]`, `getPosition()[1]`, `getPosition()[2]` returns x, y, z coordinate of tracker hit in mm.

- `const FloatVec& getCovMatrix()`
  returns covariance matrix of position coordinates. Entries are given in mm^2. The matrix elements are stored in the following order

vector index	matrix element
0	cov[x, x]
1	cov[x, y]
2	cov[x, z]
3	cov[y, y]
4	cov[y, z]
5	cov[z, z]

- `float getEnergy()`
  returns energy loss of primary particle in GeV.
- `float getTime()`
  not used for the time being.
- `int getType()`
  Type codes need agreement between subdetector groups.
- `int getQuality()`
  returns quality bits for the hit. If no bits are set, the hit has no special properties. If bits are set (the counting is such that the least significant bit has number 0), they have the following meaning:
  - bit 0: contains at least one broken channel.
  - bit 1: at least one pulse contributing to the hit is not OK.
  - bit 2: hit at the border of the pad plane.
  - bit 3 - 19: not used for the time being.
  - bit 20 - 30: reserved for private usage.
  - bit 31: not used for the time being
- `const LCObjectVec& getRawHits()`
  returns vector with objects of type `TrackerPulse` contributing to the `TrackerHit` object.

2.5 TPCHit

Obsolete class. Please use `TrackerRawData`, `TrackerData` and `TrackerPulse` instead. Future versions of LCIO will not support writing of TPCHit collections anymore.

2.6 Track

FIXME: REVIEW THIS SUBSECTION. TRACK CLASS MIGHT BE CHANGED TO AGREE WITH LC NOTE. CHECK WITH FRANK.

Objects of this type store parameters reconstructed tracks. The coordinate system, parameters and their ranges are those of [5]. This note also includes some drawings illustrating the meaning of the parameters.

- `int getType()`
  not used for time being. Bits 0-15 can be used to denote the subdetectors that have contributed hits used in the track fit. Its usage needs an agreement between subdetector groups. Bits 16-31 are used internally by LCIO.

- `float getD0()`
  returns the signed impact parameter d_0 in x-y plane in mm. This is the signed distance between the point of closest approach on the track and the reference point. The signing convention is the following: Looking from the reference point to the point of closest approach, `getD0() > 0 (< 0)` means that the particle moves from left to right (right to left).

- `float getPhi()`
  returns ϕ angle of the track (angle between track and x axis in x-y plane) in rad at ???? **FIXME: WHAT POINT?** point.

- `float getOmega()`
  returns signed curvature Ω of the track in mm^{-1}. The sign convention is such that in case of an axial magnetic field parallel to the z axis ($\vec{B} = (0, 0, B_z)$, $B_z > 0$), $\Omega > 0$ (< 0) corresponds to a particle with positive (negative) electric charge. Concerning the sign convention, please also take note of the first remark in section 3 of [5].

- `float getZ0()`
  returns signed impact parameter z_0 of the track in s-z im mm. It is the z position of the track at the point of closest approach with respect to the z coordinate of the reference point.

- `float getTanLambda()`
  returns $\tan\lambda$ which is the slope dz/ds of the straight line in the s-z plane. This parameter is related to the polar angle θ of the track by the relation $\tan\lambda = \cot\theta$.

- `const FloatVec& getCovMatrix()`
  returns covariance matrix of track parameters. The matrix elements are stored in the following order

vector index	matrix element	unit
0	cov[Ω, Ω]	mm^{-2}
1	cov[Ω, ϕ]	mm^{-1} rad
2	cov[Ω, d_0]	
3	cov[Ω, z_0]	
4	cov[Ω, tan λ]	mm^{-1}
5	cov[ϕ, ϕ]	rad^2
6	cov[ϕ, d_0]	rad mm
7	cov[ϕ, z_0]	rad mm
8	cov[ϕ, tan λ]	rad
9	cov[d_0, d_0]	mm^2
10	cov[d_0, z_0]	mm^2
11	cov[d_0, tan λ]	mm
12	cov[z_0, z_0]	mm^2
13	cov[z_0, tan λ]	mm
14	cov[tan λ, tan λ]	

- `const float* getReferencePoint()`
  `getReferencePoint()[0]`, `getReferencePoint()[1]`, `getReferencePoint()[2]` returns x, y, z coordinate of reference point in mm.

- `bool isReferencePointPCA()`
  returns true if reference point is point of closest approach.

- `float getChi2()`
  returns χ^2 of track fit (if a χ^2 fit is performed). `getChi2() < 0` indicates that a different type of fit (non-χ^2 based) has been used.

- `int getNdf()`
  returns the number of degrees of freedom of track fit.

- `float getdEdx()`
  returns average dE/dx value for track in GeV/mm.

- `float getdEdxError()`
  returns uncertainty on average dE/dx value for track in GeV/mm.

- `float getRadiusOfInnermostHit()`
  returns radius (in mm) of the innermost hit that has been used in the track fit.

- `const IntVec& getSubdetectorHitNumbers()`
  returns a vector that holds the number of hits in particular subdetectors.

- `const TrackVec& getTracks()`
  returns a vector of tracks that have been combined to this track.

- `const TrackerHitVec& getTrackerHits()`
  (optionally) returns a vector containing the hits that have been used to create this track.

3 Marlin processors

The reconstruction and analysis of the data are performed in a modular way using Marlin processors. The proposed data flow is listed in the following table

Data structure	Processor name	input/output collection name
`TrackerRawData`		TPCRawData
	TrackerRawData2DataConverter	
`TrackerData`		TPCConvertedRawData
	PedestalSubtractor	
	ChannelByChannelCorrector	
	LinearityCorrector	
	TimeShiftCorrector	
`TrackerData`		TPCData
	PulseFinder	
	ChannelMapper	
	GainCorrector	
`TrackerPulse`		TPCPulses
	HitFinder	
	HitPRFCorrector	
`TrackerHit`		TPCHits
	TrackFinder[Method]	
`Track`		TPCSeedTracks
	TrackFitter[Method]	
`Track`		TPCTracks

This table should be read in the following way: `TrackerRawData` objects are stored in a collection called "TPCRawData". These data are read and processed by the Marlin processor "TrackerRawData2DataConverter" which provides a collection of `TrackerData` objects with the name "TPCConvertedRawData". These data in turn are processed by ...

In the following the tasks of the individual processors are briefly described.

3.1 TrackerRawData2DataConverter

Processor to copy (and type convert) contents of a `TrackerRawData` object into a `TrackerData` object.

3.2 PedestalSubtractor

This module applies the pedestal subtraction for all channels using pedestals stored in LCCD.

3.3 ChannelByChannelCorrector

This processor corrects for *electronics* related differences between individual readout channels. If e. g. identical test pulses produce different pulse heights in different channels, this should be corrected for by the ChannelByChannelCorrector corrector. Variations due to varying gain should not be corrected for at this stage. It is done later by the GainCorrector. The correction factors are obtained from LCCD.

3.4 LinearityCorrector

Potential non-linear FADC behaviour is corrected by this module. **FIXME: HOW SHOULD CORRECTION PARAMETERS BE PROVIDED??? AS PROCESSOR PARAMETERS???**

3.5 TimeShiftCorrector

Time offset corrections are applied here. The offsets are retrieved from LCCD.

3.6 PulseFinder

Here the pulse finding takes place. The spreads of the pedestals needed for that are stored in LCCD. The threshold which needs to be exceeded by pulses in terms of pedestal widths is provided as processor parameter. In addition the readout frequency (stored in GEAR) is required to translate time information from FADC time bins to nanoseconds.

3.7 ChannelMapper

This module translates hardware channel IDs into software channel IDs. The mapping is retrieved from LCCD.

3.8 GainCorrector

Gain variations are corrected in this processor. The correction factors are retrieved from LCCD. Moreover a calibration constant (stored in LCCD) is required to do the charge translation from FADC counts to primary electrons.

3.9 HitFinder

The combination of `TrackerPulses` to `TrackerHits` takes place here. This requires pad geometry information (from GEAR) and knowledge about dead or noisy channels (from LCCD). Algorithmic parameters (like time window) are provided as processor parameters. To translate the drift time to the z coordinate, the drift velocity (from LCCD) is required and to calculate the energy loss of the primary particle from the charge information (in terms of primary electrons) the average energy to create an electron-ion pair (from LCCD) is needed.

3.10 HitPRFCorrector

To correct the hit positions for pad response function effects, this module is used. This requires pad geometry information (from GEAR), and gas parameters like diffusion and defocussing constant (from LCCD).

3.11 TrackFinder[Method]

This processor groups hits to tracks. The pad geometry (from GEAR) is needed for that. Algorithmic parameters (like acceptance cuts) are provided as processor parameters.

3.12 TrackFitter[Method]

The track fit is performed in this processor. Depending on the chosen fit technique, one needs pad geometry information (from GEAR) here.

4 Helper classes for LCCD storage

The LCCD helper classes provide containers to store additional conditions information. The following classes are available

- `ADCChannelMapping`
- `ChannelCorrection`
- `FieldSetting`
- `GasConditions`
- `Pedestal`

- `TPCConditions`
- `WeatherConditions`

4.1 `ADCChannelMapping`

Stores the correlation between hardware channels and software channels. Additionally the possibility exists to store the type of the electronics.

- `int getChannelID()`
  returns the hardware channel (electronics) ID.
- `int getPadID()`
  returns the software channel (pad) ID.
- `int getType()`
  returns the type of electronics used for this channel.

4.2 `ChannelCorrection`

Stores the quality of one channel as well as calibration factors and a time offset.

- `int getChannel()`
  returns the channel ID.
- `int getChannelQuality()`
  returns the status bits as an int. (Not the integer value is significant but the single bits!!)

bit	meaning if set
0	channel is not working
1	noisy channel

- `float getChannelCalibrationFactor( int Num = 0 )`
  returns a calibration factor. (More than one calibration factor can be stored.)
- `int getNChannelCalibrationFactors()`
  returns the number of stored calibration factors.
- `std::vector<float> getChannelCalibrationFactors()`
  returns a vector containing all calibration factors.
- `float getChannelTimeOffSet()`
  returns the time offset of the channel in ns.

- `bool getIsNotWorking()`
  returns true if the channel is not working.

- `bool getIsNoisy()`
  returns true if the channel is noisy.

4.3 `FieldSetting`

Stores information about all the involved voltages and fields in a GEM amplification system. This includes the drift region as well as a possible magnetic field.

- `int getNGEMs()`
  returns the number of GEMs.

- `int getGEMStatus()`
  returns the status of the GEM system.

return value	meaning
0	off
1	running normaly
2	ramping up
3	raming down
4	tripped
5	other error

- `int getDriftFieldStatus()`
  returns the status of the drift field.

return value	meaning
0	off
1	running normaly
2	ramping up
3	raming down

- `int getMagneticFieldStatus()`
  returns the status of the magnet.

return value	meaning
0	off
1	running normaly
2	ramping up
3	raming down

- `float getGEMVoltage( int GEMID )`
  returns the voltage of the given GEM. GEM 0 is the nearest to the drift region.

- `float getTransferField( int GEMID )`
  returns the field between the GEM N and N+1 in V/mm.
- `float getInductionField()`
  returns the field between the last GEM and the readout plane in V/mm.
- `float getCathodeVoltage()`
  returns the cathode voltage in V.
- `float getAnodeVoltage()`
  returns the voltage of the anode (normally a shield and the uppermost GEM electrode) in V.
- `float getDriftField( float maxDriftLegth )`
  returns the drift field in V/mm for a drift length given in mm.
- `float getMagneticField()`
  returns the magnetic field in T.

4.4 GasConditions

Stores gas related informations like gas mixture, gas temperature, gas pressure etc.

- `float getMixture( int Content )`
  returns the fraction of `Content` in the gas mixture.
- `float getTemperature()`
  returns the gas temerature in °C.
- `float getPressure()`
  returns the gas pressure in hPa.
- `float getOverPressure()`
  returns the difference between atmospheric pressure and gas pressure in hPa.
- `float getFlow()`
  returns the gas flow in l/h.
- `float getWaterContent()`
  returns the water content in the gas in ppmV.
- `float getOxigenContent()`
  returns the oxygen content in the gas in ppmV.

4.5 Pedestal

Stores informations about the pedestal of one channel.

- `int getChannel()`
  returns the channel number.
- `float getPedestalValue()`
  returns the pedestal.
- `float getPedestalWidth()`
  returns the standard deviation of the pedestal.

4.6 TPCConditions

Stores the drift velocity, diffusion coefficients and the total amplification.

- `float getDriftVelocity()`
  returns the drift velocity in mm/μs.
- `float getLongDiffusionCoef()`
  returns the diffusion coefficent for the longitudinal diffusion in $\sqrt{\text{mm}}$.
- `float getLongDefocussing()`
  returns the defocussion constant for the longitudinal diffusion in mm.
- `float getTransDiffusionCoef()`
  returns the diffusion coefficent for the transversal diffusion in $\sqrt{\text{mm}}$.
- `float getTransDefocussing()`
  returns the defocussion constant for the transversal diffusion in mm.
- `float getAmplification()`
  returns the amplification of the gas amplification device.

4.7 WeatherConditions

Stores air temperature, atmospheric pressure and relative humidity.

- `float getTemperatur()`
  returns the air temerature in °C.
- `float getPressure()`
  returns the atmospheric pressure in hPa.
- `float getRelHumidity()`
  returns the relative humidity in %.

5 Simulation specific information

FIXME: ADD TEXT

6 Non-standard readout technologies

FIXME: ADD TEXT

7 Storage of prototpye specific information

FIXME: ADD TEXT

8 Repository

FIXME: ADD TEXT

References

[1] See http://lcio.desy.de.

[2] See http://ilcsoft.desy.de/marlin.

[3] See http://ilcsoft.desy.de/gear.

[4] See http://ilcsoft.desy.de/lccd.

[5] Thomas Krämer, *Track Parameters for the Marlin Reconstruction Framework*, LC-XXX-2006-XXX **FIXME: ADD NOTE NUMBER**.